Experimenting Iterative Computations with Ordered Read-Write Locks

Pierre-Nicolas Clauss, Jens Gustedt

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Experimenting Iterative Computations with Ordered Read-Write Locks

Pierre-Nicolas Clauss
Nancy University

Jens Gustedt
INRIA Nancy – Grand Est

N° 7123

Novembre 2009

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--7123--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
NANCY - GRAND EST

Experimenting Iterative Computations with Ordered Read-Write Locks*

Pierre-Nicolas Clauss
Nancy University

Jens Gustedt
INRIA Nancy – Grand Est

Thème NUM — Systèmes numériques
Équipe-Projet AlGorille

Rapport de recherche n° 7123 — Novembre 2009 — 15 pages

Abstract: This paper presents the first experimental results of the use of our new adaptive tool for synchronization, based on *ordered read-write locks*, ORWL. They provide a new synchronizing method for data-oriented parallel algorithms and are particularly suited for iterative pipelined algorithms with out-of-core data. We conducted experiments with the classic benchmarking Livermore Kernel 23 algorithm to validate the theoretical model and measure the efficiency of the first available implementation of ORWL in the parXXL library. They show that this tool is able to efficiently control an IO bound application running on 64 parallel POSIX threads with tight data dependencies between them.

Key-words: synchronization, iterative algorithms, read-write locks, experiments

* This article is accepted for publication inhe 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, to be held in Pisa, Italy, from February 17th to 19th, 2010.

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Expérimentations de calculs itératives avec des véroux lecture/écriture ordonnes

Résumé : Ce papier présente les premiers résultats de l'utilisation de notre outil adaptatif pour la synchronisation, les véroux lecture/écriture ordonnés, ORWL. Ils fournissent une nouvelle méthode de synchronisation pour des algorithmes parallèles qui sont orientés données et en particulier ils sont bien adaptés pour des algorithmes itératives type *pipeline* sur des données dites *out-of-core*. Nous avons conduit des expériences avec l'algorithme classique de *benchmarking* "Livermore Kernel 23" pour valider le modèle théorique et pour mésurer l'efficacité de la première implantation d'ORWL dans la bibliothèque PARXXL. Ils montrent que cet outil est capable d'efficacement contrler une application qui est bornée par les ES et qui est lancée avec 64 thread POSIX en parallèle et avec des dépendances de données contrainantes.

Mots-clés : synchronisation, algorithmes itératifs, verrous lecture/écriture, expériences

1 Introduction

In many iterative computations, we can observe data dependency patterns between their different computation tasks. In this paper we handle the common case that the output of a task may be input to one or many other tasks *and* that the read and write access to that data cannot be done atomically. There are numerous examples of such dependencies for block oriented matrix computations, see [1]. Possible existing approaches to the control of such concurrent accesses are atomic snapshots, see [2, 3, 4], which focus on concurrency protection and wait-free operations. Our approach is to favor algorithmic control and data consistency, especially when using hierarchical storage. The model we developed for inter-task synchronization is conceptually independent of the execution context and can be implemented in both shared memory or distributed environments.

We called this new synchronization tool *ordered read write locks*, ORWL, see [5]. The goal of this new tool and the associated model is to provide an automated synchronization method for data-oriented parallel algorithms. The implementation of this tool is integrated as part of the parXXL library and is currently fully available for shared memory. Implementations of distributed read-write locks exist, see [6], but have not been added to the parXXL library yet. Therefore the experiments in this paper are run in shared memory to validate our model.

Furthermore, ORWL are a control structure and are thus not directly bound to the underlying data. This lets us perform experiments with out-of-core data as it is generally desirable to perform the computation on a large amount of data to get better results. The aim of this paper is to measure the efficiency of our approach experimentally, and, in particular, to investigate the possible overhead that it might impose to the application.

The basics of our model and the designs of the underlying tool for iterative parallel algorithms are given in Section 2. Then we present in Section 3 our experimental schedule and the results we obtained. Finally Section 4 concludes the paper and gives hints about future works on this tool.

2 An Adaptive Tool for Synchronization

In this section, we describe the underlying synchronization model for ORWL. All proofs on properties of the model are left to the theoretical version of this paper (see [5]).

For this model we suppose that a given set of tasks $\mathcal{T}$ is to perform some computation and that data dependencies exist between these tasks. We also suppose that tasks may be recurrent, as part of an iterative application. Data dependencies are distinguish read and write operations that are not necessarily atomic. Therefore a dependency of task v from task w is modeled by v reading data that w has written previously. Hence, v may only be executed while

w is not. Our model provides a way to control the execution order of tasks algorithmically based on their data dependencies.

2.1 Ordered Read-Write Locks

We call the building block of our model *Ordered Read-Write Locks* (ORWL), a special kind of read-write locks which have three main features:

1. A waiting queue with FIFO-policy.
2. A distinction between *post* and *acquire* operations that replace a classical one-step *lock* operation.
3. A distinction between locks (as opaque objects) and lock-handles (as user interfaces acting on locks).

All of these three features have been used previously for lock data structures, see *e.g* [7]. What to the best of our knowledge is new in our approach is their purposely combination in a single framework.

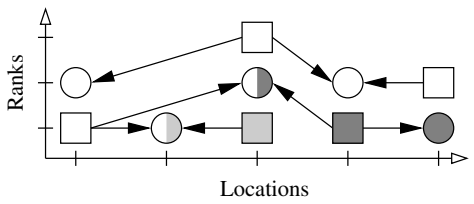


Figure 1: Sample overlay with an active task (light gray) and a delayed task (dark gray).

1. The first property ensures a controlled order of the application to its data. As we will see below the applications that we have in mind will iterate over their data, and thus we must be able to control when and what data is accessed. We also need the FIFO property of this policy to ensure and prove desirable features, such as our model of synchronization being dead-lock free, for instance.
2. The second property allows to pro-actively reserve resources. This also is in favor of iterative algorithms which require access to data in a cyclical pattern. Therefore, acquiring a lock is done by first posting a request (non-blocking operation) and later acquire the request (possibly blocking until the request can be served).
3. By doing such a pro-active locking the third property comes into play: a thread or process may define several handles (usually two in our case) on the same lock and thereby post a lock by means of one handle while still actively holding a lock via another one.

In our model, data is split in what we call a set of *data locations*, here: blocks of a matrix, and an ORWL is associated with each data location. Each task will use lock handles to post a set of inclusive lock requests (Ireq, represented as ◯) for each data location it needs to read and a set of exclusive lock requests (Xreq, represented as □) for each data location it needs to write. The set of ORWL together with the FIFO list of posted requests for data locations is called an *overlay*.

For the sake of simplicity, we will suppose in this paper that each task only posts Xreq on one single location and that each data location is only required by one Xreq (and therefore one task only). This *canonical form* can be achieved from any given set of tasks by adding auxiliary sub-tasks, see [5] for this procedure and for proofs.

Figure 1 shows a sample overlay. Data locations are placed on the horizontal axis while the FIFO order of the requests (priority rank) is placed on the vertical axis. As we suppose that the overlay is in canonical form, tasks are represented by exactly one Xreq on the data they update, with arrows to Ireq on data they need to read.

This figure shows five tasks in total, three of which already have acquired the Xreq for their data location, namely those for which the □ are found at the bottom (highest priority). The three corresponding tasks compete for non-exclusive access to the two other locations; the 'white' task shares an Ireq with the light gray task and another one with the dark gray task.

The only active task is the light gray one whose request all have the highest priority on their data locations (and thus can all be granted). The dark gray task is delayed since one request (the one after the light gray task) is still pending. Other tasks on the figure are more examples of partially and totally delayed tasks.

2.2 Recurrent Tasks

Now, to model a recurrent task of an iterative process we proceed as follows. Whenever all the lock requests that such a task has posted have been acquired, the task is said to be *active* and can perform its job. After finishing the job, a second set of lock handles allows the task to first posts copies of its requests for the next iteration. Then it releases the acquired locks to pass the control over to other tasks that operate on the same data. This strategy guarantees that all tasks iteratively get access to the data in an equitable way, see [5].

The overlay is initialized in a preprocessing phase. Once the overlay has been initialized, all tasks can compete for their requests. The order in which they get active follows the algorithmic execution scheme. In [5] we show that in the context of iterative algorithms as they are discussed here, this initialization can be done such that the subsequent computation is guaranteed to be deadlock free. We also showed that there are several ways to initialize an overlay, some of which provide the stable pattern quicker than others. In the experiments described below, we used an initialization that provides the maximal degree of parallelism immediately at the beginning of the computation.

2.3 Iterative Pipeline Algorithms

Our model is particularly suited for iterative algorithms that pipeline their data. As central example, we chose to implement the Livermore Kernel 23, which is a classic benchmark taken from LinPack, see [8]. This algorithm is usually parallelized by pipelining the computation over blocks of the initial two dimensional data matrix (starting from the upper left block down to the lower right one). It is also an iterative algorithm: the computation is repeated either for a specific number of iterations or until the computation stabilizes.

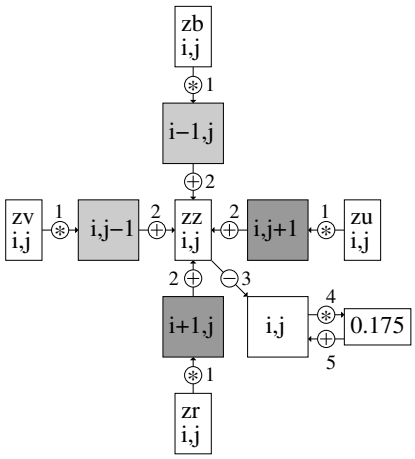


Figure 2: Local computation for the Livermore Kernel 23 benchmark on element at (i, j).

Figure 2 shows the algorithmic details for the computation of the Livermore Kernel 23 for one element of the 2-dimensional data matrix. This figure shows that there are five coefficient matrices (*zb*, *zv*, *zu*, *zr* and *zz*) which make the overall memory footprint bigger. The figure also shows the neighboring elements required to perform the computation of one element. Two of them have being previously computed (North and West, light gray), and the other two will be computed afterward (South and East, dark gray). This shows that elements on the same diagonal can be computed in parallel. The wave of computation therefore traverses the matrix $NW \rightarrow SE$.

3 Experiments

The Livermore Kernel 23 has the property that data grows quickly in memory, as it requires a set of coefficient matrices in addition to the main data matrix. In our experiments, the size of our data matrix is $16384 * 16384$ elements, each being a `double`. Thereby the size of the matrix is 2 GiB, and the total memory footprint including all coefficient matrices of an experiment is 12 GiB. This

exceeds the typical physical memory of contemporary machines and depending on the settings (in particular 32 bit machines) also the virtual memory. The layout of our experiments takes the out-of-core input into account and is set as follows:

- The data is split into blocks, each being considered as an ORWL data location.
- One POSIX thread is spawned for each data location (or block).

The scenario we followed for our experiments is the following:

1. Determine the best block size with respect to disk performance.
2. Perform the computation over 20 iterations and measure the time the application spends in the iterations for computation and for waiting.
3. Analyze the measurements.
   - Compare the amount of computation and waiting time on a per thread-basis.
   - Compare the disk bandwidth achieved with the peak performance of the disk.

	Grelon	***Capricorne***
CPU	2x Intel Xeon 5110 dual-core at 1.6 GHz	1x AMD Opteron 246 dual-core at 2.0 GHz
L2 Cache	4 MiB at 1333 MHz	1 MiB at 400 MHz
Memory	2 GiB	2 GiB
Hard Disk	80 GiB SATA	36 GiB SCSI
Age	2.5 years	4.5 years
Kernel version	2.6.26.2 (rel Aug 6, 2008)	2.6.18.6 (rel Dec 16, 2006)

Table 1: Testing machines configuration.

3.1 Data layout

For the special case of out-of-core computation, which is aimed by our experimental schedule, data is usually rearranged from the basic row-major storage layout into a more efficient block storage layout. In this layout, data is collected in blocks corresponding to the defined slicing for the application which are then

stored contiguously on disk. This allows for fast access to blocks of data during computation.

In [9], we developed an enhanced version of the block layout, which we called *optimized layout*, in which the collected blocks are stored in a special manner on disk:

1. Store the elements of the first row
2. Store the elements of the first column
3. Store the elements in the center of the block
4. Store the elements of the last column
5. Store the elements of the last row

This special layout allows for fast access to both entire blocks and single frontiers, which are required by neighbors for their own computation.

3.2 Legacy code

The implementation of the Livermore Kernel 23 has been done assuming that the use of the block layout was a pretty common situation. As ORWL are decorrelated from the computational part of an application, the goal of our model is also to allow the reuse of legacy code. Therefore, the implementation of the optimized layout has being done by reading blocks in optimized layout form into memory, then reorder them into block layout form and pass them to the original code. The writeback goes the opposite way: reorder the blocks from block layout into optimized layout and then write them to the disk.

The computational part of the application using the optimized layout thus acts as a wrapper around the computational part using the block layout, reusing most code of the original application.

3.3 Wave chaining

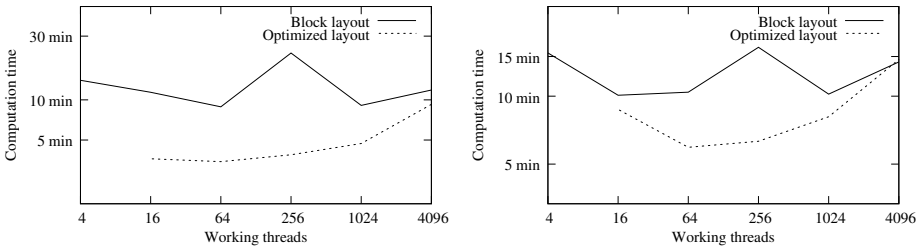
With ORWL, the synchronization footprint for each thread is limited to its neighbors. This made us observe three phases during a computation:

1. A loading phase, which occurs at the beginning of the computation and during which the hierarchical storage caches and buffers are filled with the data of the first computed blocks.
2. A stable phase, during which caches and buffers are always full due to maximal usage of the disk bandwidth.
3. A flush phase, which occurs at the end of the computation and during which caches and buffers are emptied as the computation of the last blocks finishes.

During the stable phase, the upper-left-most blocks can start a new iteration of computation while the lower-right-most blocks are finishing the current iteration. We called this phenomenon *wave chaining* and is intrinsically used by the ORWL overlay, even for out-of-core computations, without requiring manually flushing between iterations.

3.4 Results and analysis

Experiments were conducted on machines from two different clusters of the Grid'5000 platform. A summary of the configurations is given in Table 1. Experiments were initially conducted on the *Grelon* cluster, but due to availability issues, they were also run on the similar *Capricorne* cluster.



(a) On *Grelon* (b) On *Capricorne*

Figure 3: Execution time for 1 iteration and different block sizes.

Figure 3 shows the execution time for one iteration of the Livermore Kernel 23 for different (square) block sizes. This experiment is required to determine the best block size, with respect to the disk performances. On both test machines and for both layouts, best performance is obtained for $2048 * 2048$ blocks, which for our $16384 * 16384$ matrices means to decompose into 64 blocks of 32 MiB, each.

	Grelon	***Capricorne***
Sequential Input (MiB/s)	64	52
Sequential Output (MiB/s)	29	47

Table 2: Disk bandwidth performance measured with `iozone`.

.

Now, we will discuss the performance that can be expected from our setting. Therefore we measured the disk bandwidth performance with the Iozone benchmarking tool. Results for a 12 GiB file are given in Table 2.

	Grelon	***Capricorne***
Block layout	57	18
Optimized layout	40	35

Table 3: Disk bandwidth measure from computation (reads only)

Algorithmically, in the worst case the computation of one block requires reading 10 blocks (the block to update, the 5 corresponding coefficient blocks and the 4 neighboring blocks for extracting the frontiers). After the computation it then writes one block back to the disk. For the chosen case of blocks of size 32 MiB the computation of one block has to read 320 MiB and then writes 32 MiB to the disk. But as a system optimization it is possible that when the computation of a block starts, the neighboring blocks are still in memory and don't have to be re-read from disk. In the case that the system is able to obtain such a complete re-use of cached file data the actual computation reads 6 blocks (192 MiB) per computed block and writes one back.

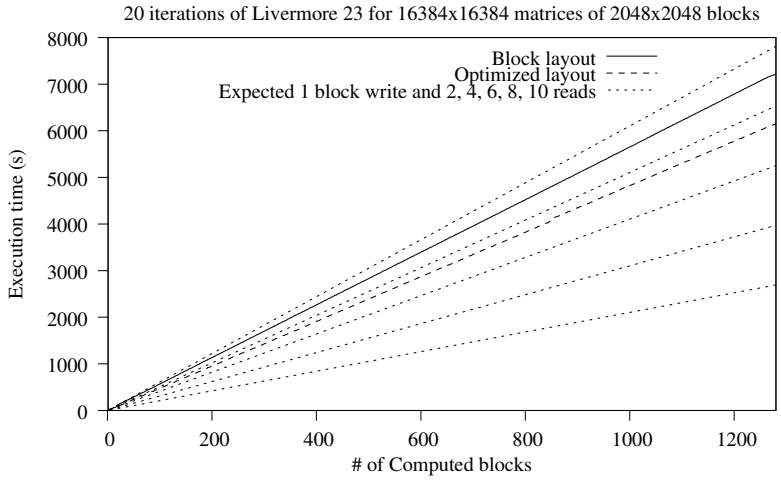
For the optimized layout, the reading footprint is similar to that later case, as it allows to read the neighboring frontiers without loading the entire blocks. So the computation of one block requires a little more than 192 MiB (6 blocks).

In total, with these reading patterns, one iteration with 64 blocks reads 20 GiB for the block layout in the worst case and 12 GiB for the case of optimal caching as well as for the optimized layout. Figure 4 shows the number of computed blocks over time for a 20 iteration run of the Livermore Kernel 23, and Table 3 reports the resulting read bandwidth.

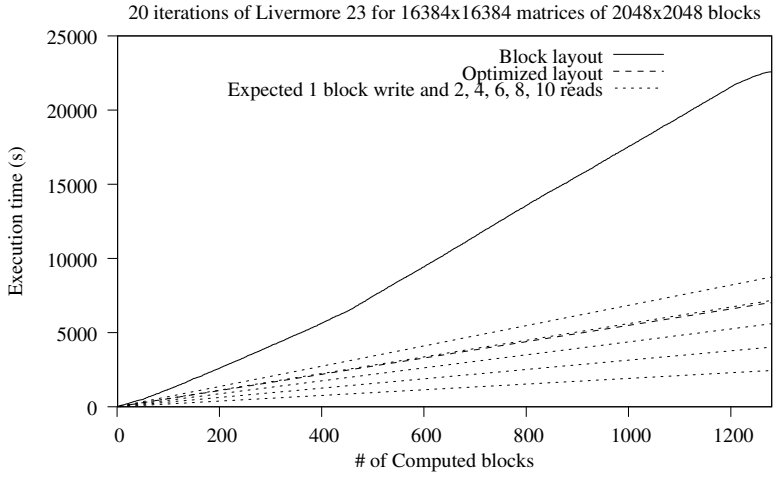
We see that the *Grelon* platform behaves as expected. The progress of the computation is linear in terms of the amount of block-computations that is handled. For the block layout, the throughput is better than the worst case and shows that the system is able to reuse some blocks from the neighbors. The overall throughput is even better for the optimized layout, which shows that this layout modification provides an important improvement.

Things are a bit different on the *Capricorne* platform. The particular case of the block layout gives poor performances. Although it is still linear, the throughput is much worse than what we would expect from the bandwidth measurements. As a cause for this lack of performance, we suspect the outdated platform software, in particular the kernel version that is used. For the optimized layout, the picture brightens a bit. Though not ideal, the throughput stays within reasonable bounds compared to what can be expected from the I/O measurements.

But in general, this experiment shows that the use of ORWL as a synchronization tool is a perfectly valid choice in out-of-core configuration and can provide a satisfying throughput compared to the measurable platform performance in terms of disk I/O.

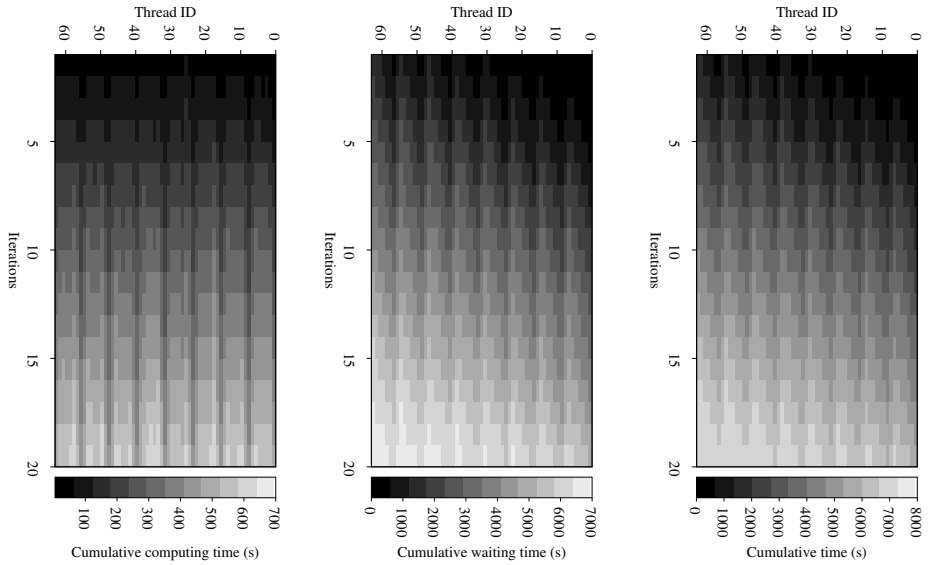


(a) On *Grelon*



(b) On *Capricorne*

Figure 4: Cumulative execution times per block for 20 iterations with both layouts.

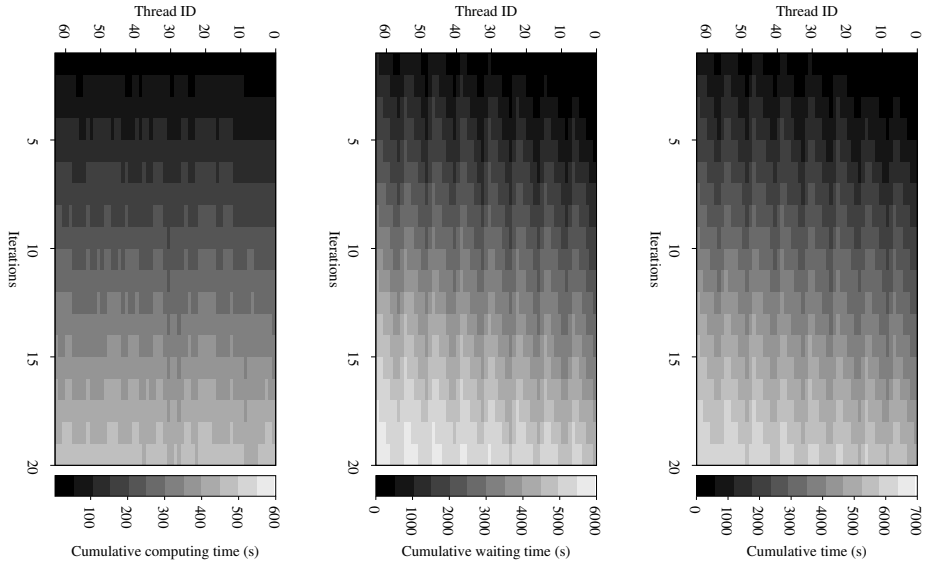


(a) Cumulative computation time per thread.

(b) Cumulative ORWL waiting time per thread.

(c) Cumulative execution time (computation + ORWL) per thread.

Figure 5: Detailed results for 20 iterations on *Grelon* with block layout.

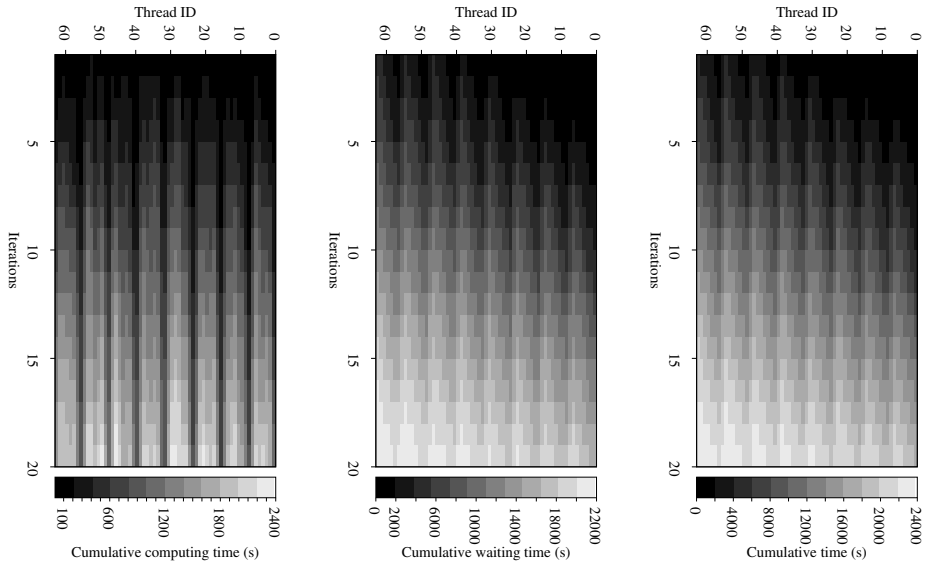


(a) Cumulative computation time per thread.

(b) Cumulative ORWL waiting time per thread.

(c) Cumulative execution time (computation + ORWL) per thread.

Figure 6: Detailed results for 20 iterations on *Grelon* with optimized layout.

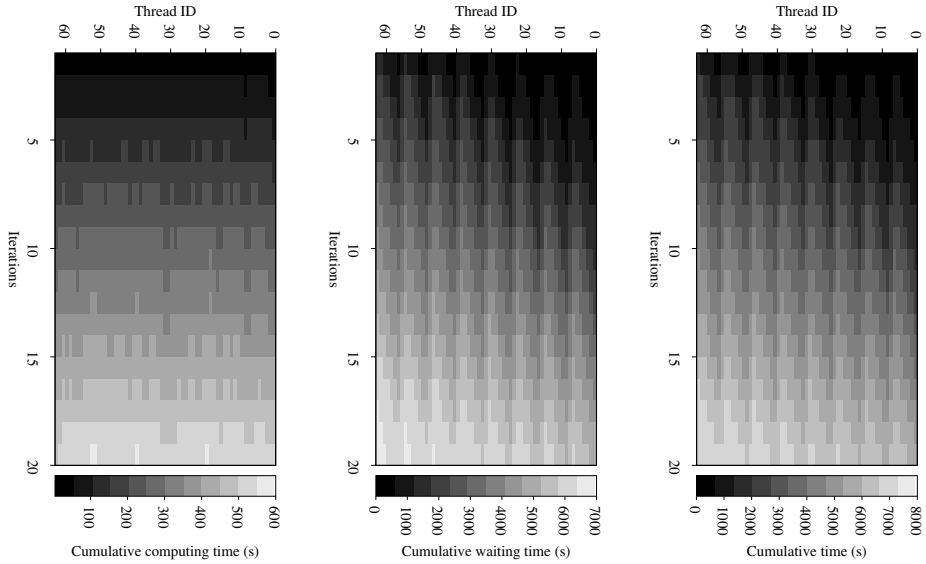


(a) Cumulative computation time per thread.

(b) Cumulative ORWL waiting time per thread.

(c) Cumulative execution time (computation + ORWL) per thread.

Figure 7: Detailed results for 20 iterations on *Capricorne* with block layout.



(a) Cumulative computation time per thread.

(b) Cumulative ORWL waiting time per thread.

(c) Cumulative execution time (computation + ORWL) per thread.

Figure 8: Detailed results for 20 iterations on *Capricorne* with optimized layout.

Figures 5, 6, 7 and 8 show the detailed timing per-thread for the 20 iteration experiment on both machines and for both layouts. These figures present three 2-dimensional grids, with the iteration on one axis and the thread ID on the other axis. The first grid gives the time spent for the computational part of the application: CPU computation and data I/O. The second grid gives the time spent waiting for ORWL requests to be satisfied. Finally, the third grid gives the sum of the first two timings. For all the grids, we divided the time scale in 12 equally spaced intervals, each filled with a plain shade of gray.

Not surprisingly, being an execution of the programs with concurrent access to processing units and data on disk, the distribution of computation time shows to be irregular. Next, we can see that the waiting time is about 10 times the computation time, and also shows some irregularities. In contrast to that, the total execution time in the third picture then is very regular. In fact, the ORWL waiting period for achieving the locks is able to absorb the irregularities that are introduced during the computation.

Also, we can notice that the general color pattern is slightly bent. This illustrates the pipeline used for parallelizing and the wave of computation that traverses the matrix from the upper-left corner (first thread ID) down to the bottom-right corner (last thread ID). We can also observe 8 horizontal ripples, each corresponding to one row of blocks. The last threads of each row start the computation one iteration later than the leading threads of the next row, hence the ripples.

Finally, we see now that the problematic executions on *Capricorne* using the block layout results in much higher irregularities for the computing time, see Figure 7(a). Indeed, the execution times of different threads are orders of magnitude apart and the systems seems to be far from equilibrium.

4 Conclusion and future work

We see from the experiments that the general execution flow is very regular for each thread. Computation progress is thus fairly distributed among the data blocks. Moreover, the usage of disk bandwidth is maximized which clearly constitutes the bottleneck for out-of-core computation. All this demonstrates that the use of ORWL and its implementation in parXXL library are highly efficient and compose well with strategies and policies deployed as part of the Linux kernel.

The experiments of this paper indicate that the overhead of the ORWL tool is negligible when compared to the amount of computation and I/O that has to be done in each iteration. Combined with the theoretical properties of the model, ORWL shows to be a very powerful synchronizing tool, for both efficient data access and ease of use by application programmers.

An implementation of ORWL for distributed environments is under construction in parXXL. We are planning to benchmark this new development under similar conditions as we have reported here. Also, we expect to conduct experiments with much larger data and real applications once this implementa-

tion has being finished and tested, to measure the footprint of communication on the overall model of synchronization.

Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadelphia, PA: SIAM, 1994.

[2] P. M. B. Vitányi and B. Awerbuch, "Atomic shared register access by asynchronous hardware (detailed abstract)," in *FOCS*. IEEE, 1986, pp. 233–243.

[3] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory," *J. ACM*, vol. 40, no. 4, pp. 873–890, 1993.

[4] P. Fatourou and N. D. Kallimanis, "Time-optimal, space-efficient single-scanner snapshots & multi-scanner snapshots using CAS," in *PODC*, I. Gupta and R. Wattenhofer, Eds. ACM, 2007, pp. 33–42.

[5] P.-N. Clauss and J. Gustedt, "Iterative computations with ordered read-write locks," *Journal of Parallel and Distributed Computing*, 2009, rR-6685. [Online]. Available: http://hal.inria.fr/inria-00330024/en/

[6] C. Wagner and F. Mueller, "Token-based read/write-locks for distributed mutual exclusion," in *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*. London, UK: Springer-Verlag, 2000, pp. 1185–1195.

[7] R. Danek and W. M. Golab, "Closing the complexity gap between fcfs mutual exclusion and mutual exclusion," in *DISC*, ser. Lecture Notes in Computer Science, G. Taubenfeld, Ed., vol. 5218. Springer, 2008, pp. 93–108.

[8] J. Dongarra, J. Bunch, C. Moler, and P. Stewart, "LINPACK," 1984, http://www.netlib.org/linpack/. [Online]. Available: http://www.netlib.org/linpack/

[9] P.-N. Clauss, J. Gustedt, and F. Suter, "Out-of-core wavefront computations with reduced synchronization," in *16th Euromicro International Conference on Parallel, Distributed and network-based Processing*, J. Bourgeois, F. Spies, and D. E. Baz, Eds. France Toulouse: IEEE, 2008, pp. 293–300. [Online]. Available: http://hal.inria.fr/inria-00176084/en/

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399